CEWES MSRC/PET TR/98-27

The BLASBench Report

by

Phillip J. Mucci
Kevin London

DoD HPC Modernization Program
Programming Environment and Training
CEWES MSRC

Work funded by the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through

Programming Environment and Training (PET)

Supported by Contract Number: DAHC 94-96-C0002
Nichols Research Corporation

Views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of Defense Position, policy, or decision unless so designated by other official documentation.

The BLASBench Report

Philip J. Mucci
Kevin S. London
mucci@cs.utk.edu
london@cs.utk.edu

March, 1998

1 Introduction

BLASBench is a benchmark designed to evaluate the performance of some kernel operations of different implementations of the BLAS or Basic Linear Algebra Subroutines. The BLAS are found in some form or another on most vendors' machines and were initially developed as part of LINPACK. Their goal was to provide a standardized API for common Vector-Vector, Vector-Matrix and Matrix-Matrix operations. A version of the BLAS is available from Netlib at `http://www.netlib.org/`. This version is subsequently referred to as the reference version. The reference BLAS are completely unoptimized Fortran codes intended as a reference for correctness to the vendors.

2 Goals of BLASBench

BLASBench aims to do the following:

- Evaluate the performance of the BLAS routines in MFLOPS/sec.
- Provide information for performance modeling of applications that make heavy use of the BLAS.
- Evaluate compiler efficiency by comparing performance of the reference BLAS versus the hand-tuned BLAS provided by the vendor.
- Validate vendor claims about the numerical performance of their processor.

- Compare against peak cache performance to establish bottleneck, memory or CPU.

3 Description

BLASBench currently benchmarks the three most common routines in the BLAS. They are:

- *AXPY* - Vector addition with scalar multiplication, $\alpha x + y$
- *GEMV* - Matrix-vector multiplication with scalar multipication, $\alpha Ax + \beta y$
- *GEMM* - Matrix-Matrix multiplication with scalar multiplication, $\alpha AB + \beta C$

The benchmark can run in either single or double precision. This is important as many systems cannot sustain the additional memory bandwidth required by using double precision data. The test space is highly tunable, as are some of the metrics BLASBench reports. The benchmark reports its results in MFLOPS/sec and MB/sec. These numbers are not computed from hardware statistics, but rather from the absolute operation and memory reference count required by each algorithm.

4 How it works

BLASBench is a C program that calls BLAS routines written in Fortran so that dynamic memory allocation can be performed. For each test, BLASBench allocates its memory in such a way that the total amount of memory taken up by each test is less than or equal to the nearest power of two. The rationale for this is that our cache sizes are always in a power of two. Once the memory is allocated, the arrays are initialized, and the test is run for a certain number of iterations. By default, the iteration count is not constant over each problem size. BLASBench trys to keep the amount of data “touched” by each run constant. This means that larger problem sizes run for fewer iterations. The effect of this is that the run time for each size is approximately constant. BLASBench provides an option to keep the iteration count constant across all tests. After each size is tested, the cache is flushed and BLASBench either proceeds to the next size or repeats that size, depending on the options given to the program. BLASBench allows you to repeat each size any number of times. This could be used to validate the numbers from each size on a time-shared system. By default, BLASBench calls the BLAS routines with the leading dimension of each array or vector set to the exact size of that vector. This means that the BLAS routines operate on the entire data set. Frequently, however, BLAS routines are called upon smaller portions of larger arrays and matrices, thus an option is provided to keep the leading dimension constant among every test. In this case, BLASBench allocates the largest possible data set, and simply changes the working set size passed to each BLAS routine.

5 Using BLASBench

5.1 Obtain the Distribution

Download the latest release from either of the following URLs:

`http://www.cs.utk.edu/~mucci/blasbench`
`ftp://cs.utk.edu/pub/mucci/blasbench.tar.gz`

Now unpack the installation using `gzip` and `tar`.

```
kiwi> gzip -dc blasbench.tar.gz | tar xvf -
kiwi> cd blasbench
kiwi>  ls
CVS/            Version         blasgraph.gp    index.html
Makefile        bb.c            conf/           make.def@
README          blasbench.html  doc/            samples/
```

5.2 Build the Distribution

First we must configure the build for our machine, operating system and BLAS libraries. All configurations support the reference BLAS if available. Running `make` with no arguments lists the possible targets.

```
kiwi> make
Please use one of the following targets:

        sunos sunos4
        solaris sunos5
        sunmp
        alpha
        linux
        hppa
        sgi
        sgi-o2k o2k
        sgi-pca pca
        t3e
        t3d
        ibm-pow2 ibm-sp2 sp2 pow2
        ibm-pow pow
```

Configure the build. Here, we are using a Solaris workstation.

```
kiwi> make solaris
ln -s conf/make.solaris make.def

Please check the VBLASLIB variable in make.def and make sure
that it is pointing to the vendor BLAS library if one exists.
Then type 'make'.
```

Examine the `make.def` file to ensure proper compiler flags and paths to the different BLAS libraries. The BLASLIB variable should contain the absolute path to the reference BLAS library and the VBLASLIB variable should contain the absolute path to the vendor's BLAS library. If one or the other is not available, just leave it blank and that specific executable will not be generated.

Now build it. Depending on whether or not both BLASLIB and VBLASLIB are set, one or two executables will be generated.

```
kiwi> make
cc -fast -dalign -DREGISTER   -c bb.c -o bb.o
if [ -f "/src/icl/LAPACK_LIBS/blas_SUN4SOL2.a" ];
  then f77  -o blasbench bb.o /src/icl/LAPACK_LIBS/blas_SUN4SOL2.a ; fi;
if [ -f "/kevlar/homes/susan/bin/solaris/libsunperf.a" ];
  then f77  -o vblasbench bb.o /kevlar/homes/susan/bin/solaris/libsunperf.a ;
fi;

Now type 'make run'.
```

5.3 Running BLASBench

BLASBench can be run from the command line, but it is intended to be run through the makefile. Running it via the makefile automates the collection and presentation process. By default, the makefile runs both executables with the arguments `-c -o -e 1 -i 1`. This says that the iteration count should be constant, the output should be reported in MFLOPS/sec, each size should be repeated only once and the iteration count should be set to one. You can change the default settings by changing the BBOPTS variable in the makefile after you have configured the distribution.

```
kiwi> make run
if [ -x blasbench ]; then blasbench  -e 1 -i 1 -c -o -v > daxpy.dat; fi
if [ -x blasbench ]; then blasbench  -e 1 -i 1 -c -o -a > dgemv.dat; fi
if [ -x blasbench ]; then blasbench  -e 1 -i 1 -c -o -t > dgemm.dat; fi
if [ -x vblasbench ]; then vblasbench  -e 1 -i 1 -c -o -v > vdaxpy.dat; fi
if [ -x vblasbench ]; then vblasbench  -e 1 -i 1 -c -o -a > vdgemv.dat; fi
if [ -x vblasbench ]; then vblasbench  -e 1 -i 1 -c -o -t > vdgemm.dat; fi
.
.
.
Now do a make datafiles.
```

At this point, depending on whether or not you have GNUplot installed on your system, you have the choice of either packaging up the datafiles for analysis on another machine, or generating the graphs in place.
First we must package up the datafiles.

```
kiwi> make datafiles
.
.
.
daxpy-kiwi.dat
dgemm-kiwi.dat
dgemv-kiwi.dat
vdaxpy-kiwi.dat
vdgemm-kiwi.dat
vdgemv-kiwi.dat
blasgraph.gp
compare.gp
custom.gp
vcustom.gp
kiwi.info

Now do a 'make graphs'.
```

```
If you don't have GNUplot, you can do this on another machine
using the kiwi-bp-datafiles.tar file.
```

Now make the graphs.

```
kiwi> make graphs
gnuplot < custom.gp > blasperf.ps
UTK BLAS graph is in blasperf.ps

gnuplot < vcustom.gp > vblasperf.ps
Vendor BLAS graph is in vblasperf.ps

gnuplot < compare.gp > compare.ps
Vendor BLAS graph is in vblasperf.ps
```

This will result in either one or three graphs. Each graph contains the performance in MFLOPS of all three operations. They are named as follows:

- blasperf.ps - Postscript file of the reference BLAS.
- vblasperf.ps - Postscript file of the vendor's BLAS.
- compare.ps - Comparison of the two.

5.4 Arguments to BLASBench

```
kiwi> blasbench -h
Usage: blasbench [-vatsco -x # -m # -e # -i #]
         -v AXPY dot product benchmark
         -a GEMV matrix-vector multiply benchmark
         -t GEMM matrix-matrix multiply benchmark
         -s Use single precision floating point data
         -c Use constant number of iterations
         -o Report Mflops/sec instead of MB/sec
         -x Number of measurements between powers of 2.
         -m Specify the log2(available physical memory)
         -e Repeat count per cache size
         -l Hold LDA and loop over sizes of square submatrices
         -d Report dimension statistics instead of bytes
```

```
        -i Maximum iteration count at smallest cache size

Default datatype   : double, 8 bytes
Default datatype   : float, 4 bytes
Defaults if to tty : -vat -x1 -m24 -e2 -i100000
Defaults if to file: -t   -x1 -m24 -e1 -i100000
```

6 Results on the CEWES MSRC Machines

The following graphs summarize our runs on each of the CEWES MSRC machines during dedicated time. The machines are the SGI Origin 2000, the IBM SP and the Cray T3E. The peak MFLOPS is as reported by the vendor and is simply computed as a product of the clock speed times the number of independent floating point multiplies and adds that can be computed per cycle. The cache size and theoretical peak MFLOPS for each machine are listed as follows.

Machine	*Cache*	*Peak*
SGI Origin 2000	32K,4MB	390
IBM SP	128K	240
Cray T3E	8K,96K	900

6.1 DAXPY (Double Precision AXPY)

$$y = \alpha x + y$$

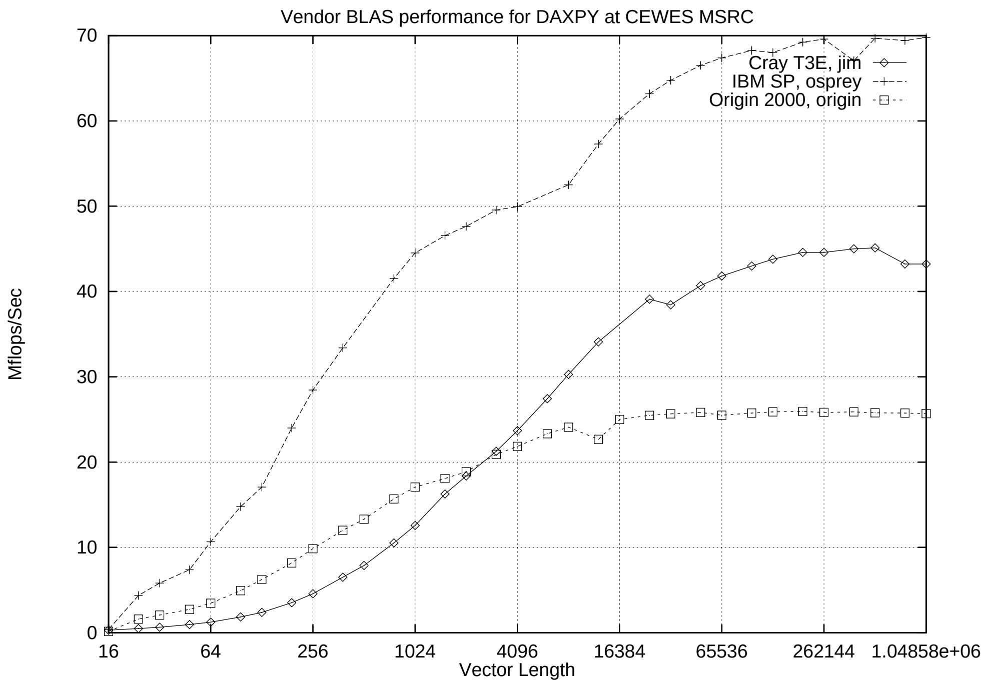

Figure 1: Performance of Vector-Vector Addition with Scale

The DAXPY operation is highly bound by the latency of cache and the throughput of the main memory subsystem. Latency is a factor because data is continuously being replaced in the cache. Every line that is loaded is used once and then discarded. The data from each vector is never *re*-used. Thus, for each cache line, we must incur the cost of flushing the dirty line and loading the new data. Because the data items are accessed sequentially, cache features like requested data first[1] line buffering and non-blocking[2] help very little.

The performance of this benchmark is highly dependent on cache line size, but independent of cache size. The reason is that the cache simply adds latency to memory accesses. As there is no data re-use, the cache is of zero benefit. The size of a cache line reflects the size of the unit of transfer between cache and main memory. Moving a lot of data at once is a performance win because of high latency of accessing the main memory.

[1]The cache controller returns the missed-upon operand first and then the rest of the line.

[2]Multiple outstanding misses can be satisfied at once.

Figure 1 shows the results of performing vector-vector addition for increasing vector lengths of double words. Here we find that the SP far outperforms the other two machines. Interestly enough, the SP has the simplest memory subsystem and the largest line size at 128 bytes. Although the T3E had the stream buffers enabled, its performance was still poor. The Origin's non-blocking cache didn't help either.

6.2 DGEMV (Double Precision GEMV)

$$C = \alpha A x + \beta y$$

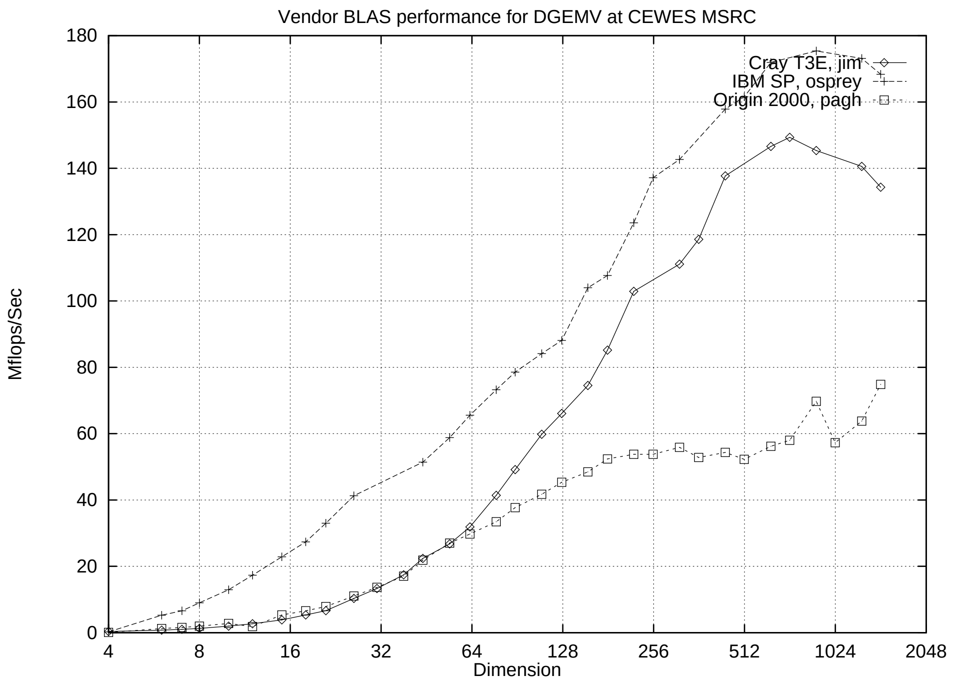

Figure 2: Performance of Matrix-Vector Multiplication with Scale

The DGEMV operation is an operation that stresses both the capacity and the latency of the cache. It provides us with some opportunity of cache re-use provided the implementation is blocked or tiled such that portions of the matrix and vectors remain in cache as long as possible.

Figure 2 shows the performance of this operation on the three machines under study. Note that while performance of the SP and T3E nearly tripled that for DAXPY, the Origin only doubled. We attribute again to its small line size of 64 bytes. The T3E experienced a large performance improvement because of its hardware prefetching. This time the prefetching is highly effective because data is re-used and the ratio of floating point operations to number of memory references is greater.

6.3 DGEMM (Double Precision GEMM)

$$C = \alpha AB + \beta C$$

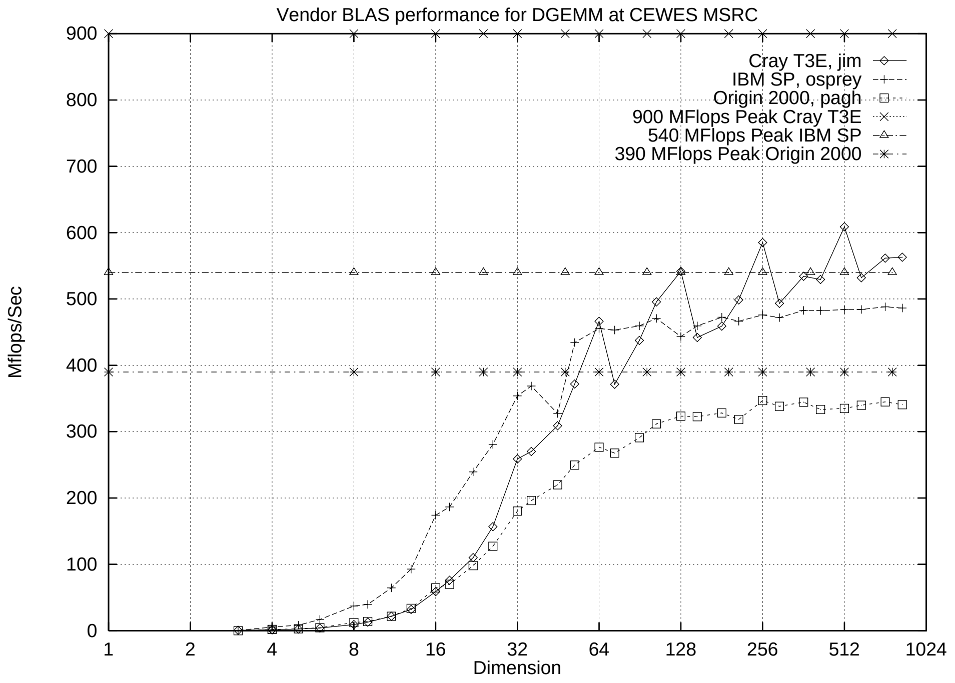

Figure 3: Performance of Matrix-Matrix Multiplication with Scale

Matrix-matrix multiplication performs well because it provides a lot of opportunity for cache re-use. By tiling the matrices, the working set can be reduced to the size of cache and thus only *capacity* misses are taken. For this reason, the performance of DGEMM has long served as a good indicator of a machine's *peak practical performance*. A machine with an adequate memory subsystem like the SP can achieve very high efficiencies, i.e. high percentage of the vendor's published MFLOP rating.

The reader should notice that clock speed and L2 cache size do not play as critical a role in the performance of this routine as one might think. The spikes in the T3E's performance curve are due to the matrix dimension being a multiple of the block size. For other cases, the GEMM routine must engage in rather lengthy cleanup code. The small cache/line size of the T3E simply exaggerates the performance loss. The SP, with its large line size and ability to issue two multiply-add instructions as well as a load/store per cycle, does quite well reaching approximately 90 percent efficiency. The Origin appears to suffer from its small cache line size and its inability to issue a load/store every cycle. Like the Power2, the R10000 processor can issue two multiply-adds per cycle, however, its memory system does not appear to be able to supply the operands fast enough for peak performance.

7 Future work

- Provide an option for measuring specific problem sizes and ranges.
- Provide an option to specify the problem sizes in dimensionality.
- Provide an option to specify the starting problem size.
- Use specialized, high-resolution timers where available.
- Add additional BLAS routines `TRSM`, `TRSV`, and `SYR2K`.
- Add parameters to tune the placement and padding of the arrays.
- Standardize configuration with GNU *autoconf*.
- Grab machine configuration and store it with each run.
- Standardize data/graph naming scheme with timestamp.

8 References

1. *Computer Architecture, A Quantitative Approach* by David A. Patterson, John L. Hennessy, David Goldberg, Published by Morgan Kaufmann Publishing, San Francisco, 1996, ISBN: 1558603298

2. *The Science of Computer Benchmarking (Software, Environments, Tools)* by Roger W. Hockney, Published by Society for Industrial and Applied Mathematics, Philadelphia, 1996, ISBN: 0898713633

The BLAS Homepage `http://www.netlib.org/blas`